

1996

## Dynamic Scheduling of Process Groups

Kuei Yu Wang

Dan C. Marinescu

Octavian F. Carbunar

Report Number:

96-084

---

Wang, Kuei Yu; Marinescu, Dan C.; and Carbunar, Octavian F., "Dynamic Scheduling of Process Groups" (1996). *Department of Computer Science Technical Reports*. Paper 1338.  
<https://docs.lib.purdue.edu/cstech/1338>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.  
Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

**DYNAMIC SCHEDULING FOR PROCESS GROUPS**

**Kuei Yu Wang  
Dan C. Marinescu  
Octavian F. Carbunar**

**Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907**

**CSD-TR 96-084  
December 1996**

# Dynamic Scheduling of Process Groups\*

Kuei Yu Wang, Dan C. Marinescu, and Octavian F. Carbutar  
Computer Sciences Department  
Purdue University  
West Lafayette, IN 47907

December 20, 1996

## Abstract

In this paper we introduce the concept of temporal locality of communication for process groups and a hierarchical decision model for dynamic scheduling of process groups. Empirical evidence suggests that, once a member of a process group starts to communicate with other processes in the group, it will continue to do so, while an independent process will maintain its state of isolation for some time. Other instances of inertial behavior of programs are known. Temporal and spatial locality of reference are example of inertial behavior of programs, exploited by hierarchical storage systems; once a block of information (program or data) is brought into faster storage, it is very likely that it will be referenced again within a short time frame. When process groups exhibit temporal locality of communication, this information can be used to hide the latency of paging and I/O operations, to perform dynamic scheduling to reduce processor fragmentation, and to identify optimal instances of time for check-pointing of process groups. In our scheduling model the supervisory process of a process group collects information about the dynamics of the group and shares it with local and global scheduling agents.

---

\*Work supported in part by NSF grants BIR-9301210 and MCR-9527131, by the Scalable I/O Initiative, by a grant from Intel Corporation and by CNPq Brazil

# Contents

1	Overview	2
2	A hierarchical decision making model for dynamic scheduling	4
3	Exploiting temporal locality of communication for dynamic scheduling	7
4	Empirical evidence supporting temporal locality of communication	10
4.1	Correlation of Communication Events . . . . .	10
4.2	The applications . . . . .	13
4.3	Experimental results . . . . .	15
4.4	Discussion of the results . . . . .	22
5	Conclusions	23
6	Acknowledgments	24

## 1 Overview

The availability and usage of multiprocessor systems have grown rapidly. Multiprogramming parallel machines, with several parallel applications being processed concurrently, has been proposed as a method to improve utilization of multiprocessor systems.

Multiprogramming can be achieved by time sharing a multiprocessor system among several applications, as in the uniprocessor systems, or by space sharing where subsets of processors are assigned to various applications. Although multiprogramming allows better service to be provided to the users, it also complicates the processor allocation issues in multiprocessor systems.

There are two classes of scheduling algorithms for multiprogrammed parallel systems static and dynamic, [3],[8],[9], [11], [12]. The *static scheduling algorithms* are non-preemptive, each application runs to completion without interruption on the set of processors initially allocated for it. The *dynamic scheduling algorithms* are able to adjust the spatial and the temporal allocations of processing elements to the needs of the parallel applications and at

the same time attempt to optimize the overall use of system resources.

The static scheduling algorithms are simpler to implement and have lower overhead than dynamic algorithms. Static scheduling algorithms are capable in principle to support a mix of sequential and parallel computations with various degree of parallelism but rather inefficiently due to *processor fragmentation*. Dynamic scheduling algorithms are rarely used because it is very difficult to characterize analytically the behavior of interacting processes. The interaction between the computation time of the processes, the time spent while waiting for messages from other processes, and the time spent while waiting for processor to be available are too complex to be predicted accurately.

We now introduce the basic terminology used throughout the paper. A *process group*,  $P = \{P_1, P_2, \dots, P_i, \dots, P_n\}$ , consists of  $n$  processes,  $P_i$ ,  $i = 1, n$ , that need to be scheduled concurrently on the *Processing Elements*, *PEs*, of a parallel system because they communicate with one another.

The corresponding scheduling mechanism called *gang scheduling* requires concurrent scheduling of all members of a process group regardless of the dynamics of their synchronization. When a member  $P_i$  of a process group  $P$  encounters a page fault, in case of demand paging, or waits for the completion of an I/O operation, the process blocks, but the local scheduler does not perform a context switch. Gang scheduling with busy waiting, is used extensively by existing MIMD systems, e.g., Paragon, CM5, SP2, Alliant FX/8 for parallel applications which exhibit fine-grain interactions. When combined with support for demand paging, this strategy leads to wasted CPU cycles and longer execution time [2], [13].

*Co-scheduling* is an alternative to gang scheduling [1], [10]. In case of co-scheduling, the dynamics of synchronization requirements of the application is taken into account; only those members of the process group which need to communicate with one another are scheduled concurrently.

Determining the *working set of the process group* – the set of members which need to communicate with one another at a given moment of time – is a difficult proposition, and we are not aware of any system which implements such a policy. Another practical motivation for gang scheduling with busy waiting are memory limitations, often the amount of physical memory available to individual PEs is insufficient to accommodate multiple process groups. Yet, gang scheduling with busy waiting is likely to incur increasingly higher costs because the processor speed experiences a considerable higher improvement rate than the I/O speed. Moreover, one way to increase the I/O bandwidth is to coalesce I/O operations based on temporal and spatial locality of reference, to transfer larger blocks of data. This approach generally leads to an increase of the time a process is blocked waiting the completion of the

I/O operation.

We believe that there are compelling reasons to study scheduling of a mix of sequential and parallel computations with various degrees of parallelism. Many applications involve processing of large volumes of data and they need computing resources well beyond those provided by a single processor system. There are classes of applications which grow up naturally, they require modest computing resources during early stages of the computation and increasingly larger ones as computation progresses. The structural biology codes discussed in Section 4 of this paper belong to this class.

As the cost of microprocessors drops and faster and less costly interconnection networks become available, more systems ranging from PCs, to workstations and to servers are built as multiprocessor systems. Such systems can in turn be interconnected using ATM technology and thus create clusters capable to provide computing resources necessary to solve all but a few very large problems. For very large problems massively parallel systems and static partitioning strategies may still be the only alternative.

Maurice Wilkes notes in his essays [16] "it would be too much to expect that a single operating system can be found that would be suitable both for powerful workstations and also for laptops or even smaller computers.... we may hope for a range of operating systems that were, in some sense, compatible or at least friendly to one another." At the present time there is a wide gap between systems supporting execution of individual processes and those supporting concurrent execution of dependent processes. The very moment an application crosses the boundary between sequential and concurrent execution it moves into unfriendly environments.

The contributions of this paper are: a hierarchical decision making model for a system capable to support execution of both individual processes and process groups, and the concept of temporal locality of communication amongst the members of a process group. We believe that empirical studies of parallel programs are essential to understand behavioral patterns and we report measurements on communication activity of parallel programs. The phase structure of parallel programs has been known for quite some time, therefore temporal locality of communication is not totally unexpected. Hierarchical decision making schemes are common for the control of complex systems.

The paper is organized as follows. Section 2 presents a model for hierarchical decision making, involving local schedulers, supervisory processes attached to process groups and global schedulers. In Section 3 we discuss the concept of temporal locality of communication and describe a simple mechanism to hide latency of page faults and I/O operations for temporarily independent processes. Then we present a simple state transition diagram

used to detect at run time when a member of a process group is temporarily independent and then show how dynamic scheduling can be used to minimize the effects of processor fragmentation. Section 4 presents the empirical evidence suggesting that process groups exhibit temporal locality of communication. We describe the programs we have instrumented and the clustering algorithms used to identify the clusters of communicating processes in a process group.

## 2 A hierarchical decision making model for dynamic scheduling

There is a wide gap between the design philosophy of sequential and parallel systems with important consequences upon the design principles of parallel applications. Sequential systems perform a variety of resource management and optimizations on behalf of applications, for example memory management including demand paging, I/O buffering, caching and others. The architects of parallel systems take the view that the application is responsible for managing resources allocated to it. Load balancing along with checkpointing, I/O optimization, and possibly others, are functions expected to be built into an application.

Sequential system schedulers attempt to balance processor utilization and the turnaround time and support sharing of most system resources among competing tasks. Parallel systems schedulers dedicate most, or all resources to one task at a time, with the objective of reducing its total execution time.

If we accept the economic realities of today we need to narrow the gap between the design philosophy of sequential and parallel systems and at the same time reexamine some of the techniques used to build parallel programs. Efforts to optimize the I/O performance of parallel systems by coalescing I/O requests, to support demand paging and automatic checkpointing are under way. At the same time we need to rethink the way communication libraries are integrated into a system. Schedulers and supervisory processes in the system need to be informed when collective communication takes place.

Rather than forcing the designer of an application to address the load balancing problem it would be more beneficial to request her to provide information concerning the problem size and hints on how to partition the work and let the system attempt load balancing across multiple process groups in a dynamic scheduling environment. User built-in load balancing strategies and dynamic scheduling are unlikely to mix well together. A similar dilemma existed when virtual memory was introduced, previous techniques based upon program seg-

mentation and overlays used to fit large programs into smaller size memories had to be abandoned in favor of demand paging.

In the followings we sketch a model of a system with dynamic scheduling based upon a hierarchical decision making process with several levels of agents able to exchange information among themselves. *Local schedulers* are responsible for short term decisions concerning processor allocation. They perform context switching of a mix of single processes and members of process groups. A local scheduler uses additional information provided by higher level agents to treat processes belonging to a process group. A member of a process group may not be dispatchable even if it is ready to run, if it has reached a global synchronization point and there are other members of the group yet to reach that point. By the same token if such a process experiences a page fault while in the midst of a global exchange with other processes in the group the local scheduler should not force a context switch but allow the process to continue until the global exchange comes to an end.

To ensure the coordination functions of a process group the system creates *supervisory processes* for each group at the time when the application is started. Supervisory processes run at higher priority levels than other processes of any group. The functions of a supervisory process are: (a) process directives and hints, (b) negotiate with global system schedulers allocation of resources needed by the process group, (c) interact with local schedulers of the processors assigned to the application, (d) gather information about the communication patterns of the application, about collective communication operations, about global synchronization by interacting with the communication libraries, (e) intercept I/O requests of the members of the group and optimize I/O e.g by coalescing I/O operations, (f) perform automatic checkpointing, and (g) implement some form of fault tolerance.

To avoid the dangers of a single point of failure supervisory processes may need to be replicated. This solution could be very complex and costly and should only be considered when the failure rates are extremely high. An alternative, suitable for most systems, is to have the global schedulers check and restart a supervisory process in case of failure.

Some applications like the ones described in this paper, are built around *self-scheduling algorithms*. Such algorithms discover at run time the size of the problem and the number of processors in the partition and attempt to divide the work as evenly as possible amongst the processors. This idea can be extended and we believe that with adequate hints, the supervisory process of a process group can perform dynamic load distribution and reassignment of unfinished work in case of the failure of one of the processors assigned to the group.

The supervisory process could also include algorithms to discover patterns and temporal correlations, like the ones described in [13] or connected components algorithms like the one



used for data analysis in this paper. Another possibility is to include learning modules. When the application is executed repeatedly the supervisory process may collect trace data during early runs and then use these data for resource optimization during later runs.

Another group of scheduling agents are *global schedulers*, agents responsible for global decisions and long term allocation of system resources to process groups. When systems are interconnected, global schedulers coordinate their decisions.

We expect an increase of the system overhead in a system supporting dynamic scheduling of process groups due to the added complexity of global decisions. Gathering the information necessary to make scheduling decisions can be very costly. For example detecting that two process groups compete for a partition and none is able to make substantial progress unless the other is suspended requires a considerable amount of information to propagate from the low level agents up. There are also operations with relatively low overhead. For example detecting that a process group has reached a barrier synchronization point or is about to enter a collective communication phase requires only that the message passing library passes the information to the supervisor process of the group. Another low cost scheme for gathering scheduling information is discussed in the next section. We argue that local schedulers can detect with ease if a process is temporarily independent and exploit this information to minimize the number of cycles wasted by busy waiting. Then we discuss more complex schemes involving several scheduling agents.

### 3 Exploiting temporal locality of communication for dynamic scheduling

First we discuss means to hide the high latency of I/O operations. The solution is based upon our studies of the dynamics of synchronization for several applications we have examined. For simplicity, assume that time is slotted and the duration of a slot,  $\Delta$ , is much larger than the time required to perform a local context switch of process  $P_i$  running on  $PE_k$ . We consider  $\Delta$  to be approximately equal to the time to perform an I/O operation. Let us for the moment assume that we can predict with some level of confidence that a member of a process group is unlikely to experience a communication event during the next few slots. Call such a process a *temporarily independent* process, or TI-process. If at time  $t$ , process  $P_i \in P$  experiences an event leading to blocking, then one could consider the following alternative to busy waiting:

- (a) Determine if  $P_i$  is a TI-process and, if so, find if there is another TI-process  $Q_j \in Q$  ready to run on  $PE_k$ , where  $P_i$  is the currently running process. This is a local decision of the scheduler of  $PE_k$ .
- (b) If conditions in (a) are satisfied then perform a local context switch and let  $Q_j$  run until  $P_i$  becomes dispatchable again. Otherwise, block  $P_i$  and wait until it is ready.

The strategy described above is practical if one can determine with relative ease when a member of a process group belongs to the TI-class.

To explain the intuition behind the heuristics used to determine if a process belongs to the TI-class, we review briefly the concept of locality of reference, another example of inertial behavior of a program. Modern computer systems have a hierarchy of storage, primary and L2 caches, main memory, virtual memory based upon demand paging. As one traverses this hierarchy, the latency increases, but the cost of storage decreases, and the size of the available storage increases. Hierarchical storage systems work because programs tend to exhibit both temporal and spatial locality of reference. This implies that once a block of information (program or data) is brought into faster storage, it is very likely that it will be referenced again within a short time frame.

The question we address is if it is reasonable to assume that the synchronization requirements of a member of a process group exhibit the same type of inertial behavior; namely, if a process which has been communicating during the immediate past will continue to do so during the next slots, and conversely, if it has not been communicating, it will continue to work in isolation in the immediate future. If we can support a positive answer to this question, then we can design very simple algorithms to determine if at time  $t$  process  $P_i \in P$  belongs to the TI-class or not. The analogy with demand paging can be extended, and we can use as models the page replacement algorithms.

We now examine more complex situations which require cooperation among scheduling agents at different levels. Given a process group  $P$  of size  $n_P$ , let us define two  $n_P$  bit vectors:  $PGC_P$ , Process Group Communication status vector, and  $PGTI_P$ , Process Group Temporarily Independent status vector. If  $P_i \in P$  belongs to the TI-class, then  $PGC_P(i) = 0$  and  $PGTI_P(i) = 1$ ; if  $P_i$  does not belong to the TI-class,  $PGC_P(i) = 1$  and  $PGTI_P(i) = 0$ .

Call  $n_P^C(t_k)$  the number of processes  $P_i \in P$  with the PGC bit on at time  $t_k = k \cdot \Delta$  and  $n_P^{TI}(t_k)$  the number of processes in  $P$  belonging to the TI-class. In general,  $n_P^C(t_k) + n_P^{TI}(t_k) \leq n_P$ . Indeed, if the state of  $P_i \in P$  is given by the tuple  $(PGC_P(i), PGTI_P(i))$  a process may be in an transient state  $(0,0)$ . The full state transition diagram of process  $P_i$  is given in Figure 1.

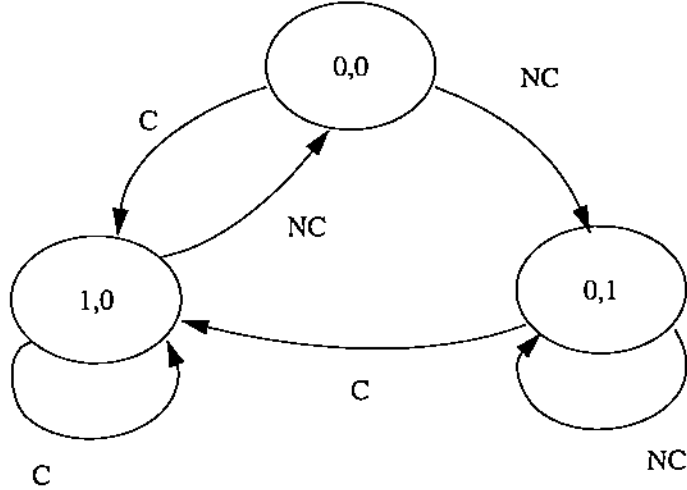


Figure 1. The state transition diagram of process  $P_i \in P$  at time  $t_k$ . The time is slotted  $t_k = k \cdot \Delta$ . A transition in slot  $k$  is determined by the occurrence of a communication event (C) or the absence of a communication event (NC). Initially, processes start in state (0,1).

We conjecture that knowing the pair of vectors PGC and PGTI for a set of process groups  $P, Q, R, \dots$ , a global scheduler could make scheduling decisions leading to a better utilization of the resources of a system with  $N$  processing elements. One of the main drawbacks of static scheduling is *processor fragmentation*. If we call  $n_P, n_Q, n_R$ , etc., the number of processes in each of the process groups  $P, Q, R$  and so on, a static scheduler attempts to partition the system into disjoint partitions and allocate each process group to one partition, such that  $n_t = n_P + n_Q + n_R + \dots$  be as close as possible to  $N$ . But in this scheme,  $N - n_t$  processing elements may end up not assigned to any partition, leading to the so-called processor fragmentation. Knowing the dynamics of synchronization for each process group provides some flexibility. Instead of being constrained to schedule all  $n_P$  processes belonging to process group  $P$ , the global scheduler may elect to schedule any number  $n_P^a$  of members of  $P$  such that  $n_P^c \leq n_P^a \leq n_P$ . Clearly  $n_t^a = n_P^a + n_Q^a + n_R^a + \dots$  may cover  $N$  better than  $n_t$  previously defined.

Whenever  $n_P^c$  increases because one or more processes leave their state of temporal isolation, the global scheduler needs only to discover which process group has more active processes than its current communication vector and order few local context switches.

Checkpointing is a problem of concern for massively parallel systems, yet few systems support automatic checkpointing. We contend that, knowing the two state vectors defined earlier in this section, the supervisor of a process group could make intelligent decisions when

to carry out checkpointing for the group.

Checkpointing a process group at a time of intense communication, when  $n_P^C$  is close to  $n_P$ , involves a considerable overhead; there are  $n_P^a (\simeq n_P^C)$  active processes' contexts to be saved and all the active pages of all  $n_P^a$  processes to be backed up. An alternative strategy is doing the checkpointing when  $n_P^a \simeq n_P^C \ll n_P$ ; that is, when the only active processes are those that communicate with one another. The strategy can be achieved by following the TI-status vector and, when  $n_P^{TI}$  exceeds a certain threshold, the local schedulers force context switching for all processes  $P_i \in P$  which belong to the TI-class, at the same time, checkpointing processes being released. When only communicating processes in the process group are active, the entire group is propitious for checkpointing.

## 4 Empirical evidence supporting temporal locality of communication

To gather experimental evidence to confirm temporal locality of communication among members of a process group, we have monitored several parallel applications, [15]. We have instrumented the communication statements and examined the collected data. One possible approach to represent the results is to display the time elapsed between successive communication events for every thread of control. Such a representation has obvious disadvantages since it is dependent upon the number of threads of control. We have opted to represent the dynamics of the size of the working set of the process group. If we consider a window of size  $\Delta$ , for each  $\Delta$  the synchronization dependencies partition the entire process group  $P$  into disjoint working sets. We represent the size of the largest working set as a function of time.

This approach does not capture the communication patterns of processes communicating at a given time. It is unable to distinguish between symmetric versus asymmetric communication patterns or recognize that there may be more than one group of size equal to the working set, or that the working set may consist of several tightly or loosely coupled subgroups. All these are important issues that need to be addressed by further studies of the communication activity of parallel programs. They may provide more hints for optimizing the program execution. The goal of our study is to find out if temporal locality of communication inferred from the multi-phase structure of parallel programs could indeed be observed in practice.

More elegant and possibly more efficient algorithms for connected components than the one we describe in Section 4.1 as part of our data analysis methodology are likely to exist.

## 4.1 Correlation of Communication Events

For the characterization of communication patterns, we want to identify sub-groups of processes within a process group which are related to each other through communication events. To clarify the definitions, we use the following notation:

- $A$  = a process group;  $A = \{A_1, A_2, \dots, A_{n_A}\}$ .  
 $A$  consists of disjoint sub-groups of communicating processes:  
 $A = \{G_1 \cup G_2 \cup \dots \cup G_{n_G}\}$ , for  $i \neq j, G_i \cap G_j = \emptyset$
- $n_A$  = the size of the process group.
- $n_G$  = the number of clusters (sub-groups).
- $G_i$  = cluster  $i$ .
- $g_i$  = the number of elements of cluster  $i$ .

Following the definition of working set model introduced by Denning [7], we define the communication working set of process  $A_k$  at time  $t$  as:

- $C_{A_k}(t, \Delta)$  = collection of processes that communicate with process  $A_k$  during the time interval  $(t - \Delta, t)$
- $\Delta$  = working set parameter

For a given time window  $[t - \Delta, t]$ ,  $C_{A_k}(t, \Delta)$  can be obtained by looking at the communication events between process  $A_k$  and others of the same application during the time window. For example,  $C_{A_k}(t, \Delta)$  can be calculated from the set of trace files containing the communication events of all processes of a parallel application.

Based on the assumption that non-overlapping communicating groups may be scheduled independently, our objective is to identify sub-groups  $\{G_1, G_2, \dots, G_{n_G}\}$  based on  $C(t, \Delta)$  of each process, obtained from the trace data of program execution.

Given a set of trace files of a parallel program  $A$  executed on  $n_A$  PEs with the overall execution time  $T_A$ , we are interested in the size and the elements of the largest communicating sub-group throughout the execution from  $T_0$  to  $T_A$ , at each interval  $\Delta$ .

Let  $M$  be an  $n_A \times n_A$  matrix, a communication relationship map, where each element  $M[i, j]$  ( $0 \leq i, j < n_A$ ) can be either one or zero defined as follows:

---

```

t = T0
while t < Tp {
  for each trace file k {
    select trace records belonging to [t - Δ, t] ;
    construct CAk (communication relationship map, fill M[k, ]) ;
  }
  use M to find the largest cluster of communicating processes ;
  output the elements and the size of the largest cluster ;
  t = t + Δ ;
}

```

Figure 2. Algorithm for calculating the largest communicating cluster (sub-group).

---

$$M[i, j] = \begin{cases} 1 & A_i \text{ sends messages to } A_j \text{ during the time window } \Delta \\ 0 & \text{if there is no communication between } A_i \text{ and } A_j \end{cases}$$

The pseudocode for determining the largest communication working set is illustrated in Figure 2. The algorithm for finding communicating sub-groups (or clusters) is similar to the algorithm in [4] for determining the connected components of an undirected graph using the disjoint-set data structure. Figure 3 adapted from [4], illustrates the pseudo code for finding connected components (clusters).

---

```

for each  $A_i \in A$  {
  do
    Make_set( $A_i$ )           // create a set for each  $A_i$ 
}
for each  $M[i, j] = 1$  {
  do
    if Find_set[i]  $\neq$  Find_set[j] then
      Union(i, j)           // unite two disjoint sets
}

```

Figure 3. Procedure that uses disjoint set operations for finding connected components (clusters).

---

Using the *union by rank* and *path compression* heuristics presented in [4], the running time for finding connected components is almost linear in the total number of operations. (Make\_set, Find\_set and Union operations.) A snapshot of the computation is illustrated in Figure 4. A directed graph with four connected components illustrates the *send* communication relationship among components, Figure 4(a). Note that, although we have a directed graph, the edges can be treated as undirected edges without altering the clustering algorithm. Figure 4(b) illustrates the communication matrix  $M[i, j]$  and the collection of clusters  $\{G_1, G_2, \dots, G_4\}$  after processing  $C_{A_k}$  of each PE  $A_k$ .

## 4.2 The applications

We discuss below two programs in the Molecular Replacement suite we have examined, *envelope* and *fftsynth*, and results gathered during their execution on a Paragon system [6].

The *envelope* program computes the molecular envelope of a virus. It needs as input a 3-D lattice with up to  $10^9$  grid points and produces a lattice of equal size as output. For every grid point, information about the electron density and a mask describing if the grid point is located in the protein, nucleic acid or solvent is provided. A spherical virus has icosahedral symmetry, and the program exploits this symmetry to get better estimates of the electron density at every grid point by calculating the average of the electron density of all points related by non-crystallographic symmetry. The program implements a shared

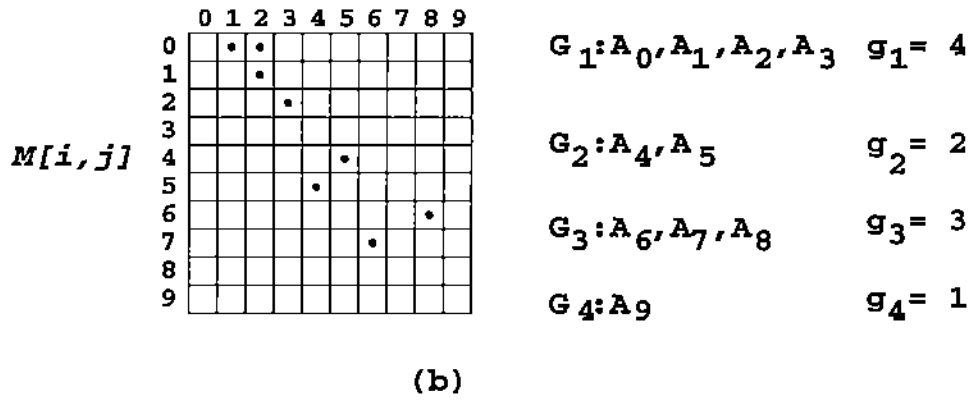
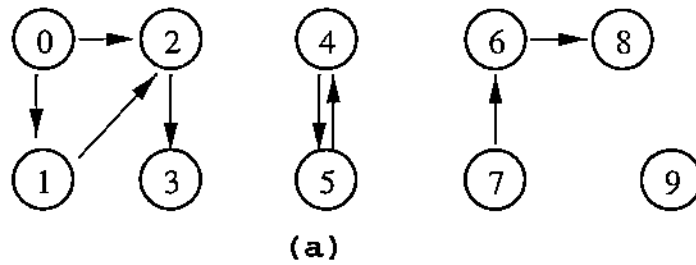


Figure 4. Snapshot of communication working set at  $t_k = k \cdot \Delta$ . There are four clusters,  $G_1$ ,  $G_2$ ,  $G_3$  and  $G_4$ . The size of the largest cluster is 4. The algorithm indicates that 9 processes (0 to 8) are communicating and one is temporary independent.



virtual memory and operates in two modes, the DD mode, where the shared virtual memory resides on the external storage device, and the DC mode, where the input data is distributed over the set of compute nodes. Reference [5] describes different data management strategies for implementing a shared virtual memory. The entire data set is partitioned into *Data Allocation Units*, DAUs. The *working set* of DAU<sup>j</sup> consists of all DAUs needed to carry out the computations associated with DAU<sup>j</sup>. DAU *faults* occur when a compute node needs to access a DAU stored elsewhere; the penalty for a DAU fault can be significant. In the DD mode a DAU fault requires a disk access, and in the DC mode it requires access to data stored on a different compute node or on a data server node. A load balancing algorithm distributes the DAUs based upon an estimate of the amount of work associated with each of them. Both modes exploit the locality of reference and attempt to minimize the number of DAU faults by processing the DAUs assigned to each compute node to maximize the intersection of the working sets of DAUs processed in sequence.

The second program, `fftsynth`, carries out a 3-D FFT. It reads in a set of complex valued structure factors (discrete Fourier coefficients), computes the FFT and writes out the calculated electron density. A 3-D FFT is obtained by a 2-D FFT followed by a 1-D FFT in the third dimension. The algorithm requires a global exchange between phase one and phase two of the algorithm. If the amount of the combined local memory of the PEs is large enough to hold all the data, then intermediate results are exchanged through message passing among PEs, otherwise the global exchange is done using an external file.

### 4.3 Experimental results

We have instrumented all the communication statements of the `envelope` and `fftsynth` program. The analysis package processes the trace records and calculates the sets of communicating sub-groups for each time window of size  $\Delta$ . The output of the analysis is the working set profile of a parallel application showing the largest process working set size for each time window  $\Delta$ .

For example, Figures 5, 6 and 7 show the communication working set of the `envelope` program running on 64 nodes, with  $\Delta$  equal to 0.2, 0.5 and 1 second, respectively. Note that these graphs derive from the same set of trace record files, but are processed with different  $\Delta$ .

For an execution of a parallel application using  $n_A$  processes, the maximum communication working set of size  $n_A$  indicates that each of the  $n_A$  processes is directly or indirectly related to all others during the interval  $[t - \Delta, t]$ . On the other hand, a working set size equal

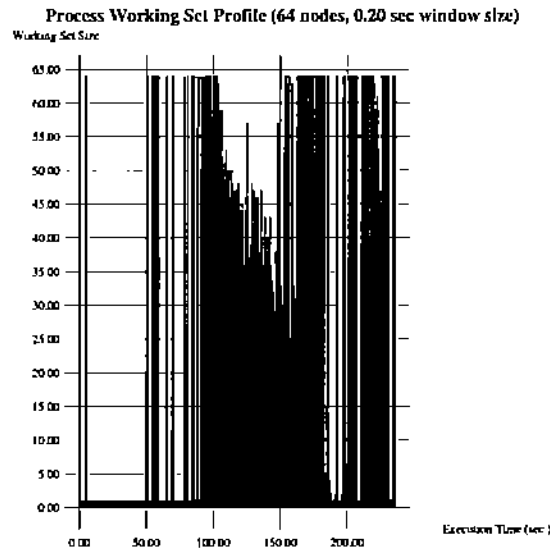


Figure 5. Envelope working set profile,  $\Delta = 0.2$  seconds, for a parallel execution using 64 nodes.

to one indicates that none of the processes communicates with others during the period.

The parameter  $\Delta$  plays an important role in the analysis of the processor working set. The characterization of the communication groups could provide information for a dynamic processor scheduling algorithm to make better use of computing resources. For example, by using  $\Delta$  in the order of magnitude of the time necessary for a group context switch, we could observe the communication dependency among processes of a process group within the granularity of context switch time.

Figures 8, 9, and 10 show the processor working set of the envelope program running on 32 nodes for  $\Delta$  equal to 0.2, 0.5 and 1 second respectively. From both sets of graphs, we note that the envelope program presents a strong communication relationship during most of its execution time (for example, the period between 80 seconds and about 170 seconds in the 64 node execution and the period between 90 seconds and 240 seconds in the 32 node execution).

The `fftsynth` program presents a communication characterization completely different from the envelope program. Even when the global exchange is carried out "in-place", as shown in Figures 11, 12, and 13 there is little communication dependency during the execution of the `fftsynth`. Figures 14, 15 and 16 show the working set profile of a `fftsynth` execution using external storage for the global exchange.

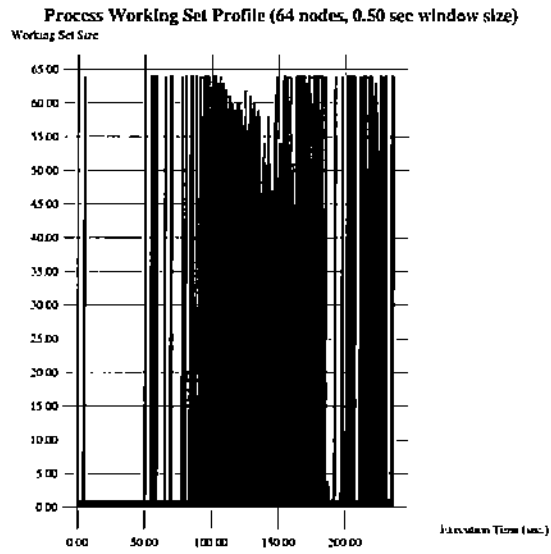


Figure 6. Envelope working set profile,  $\Delta = 0.5$  seconds, for a parallel execution using 64 nodes.

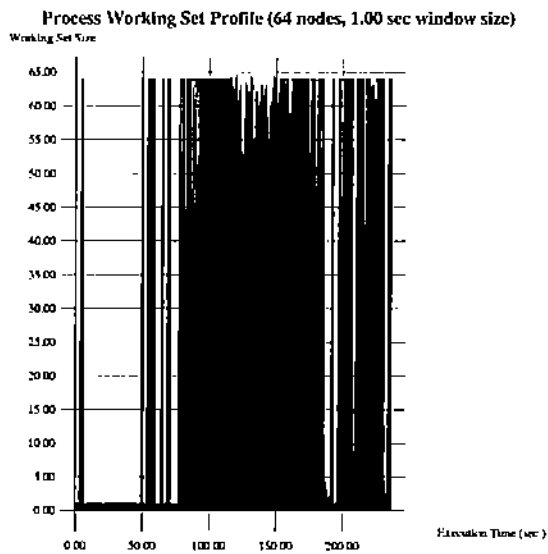


Figure 7. Envelope working set profile,  $\Delta = 1$  second, for a parallel execution using 64 nodes.

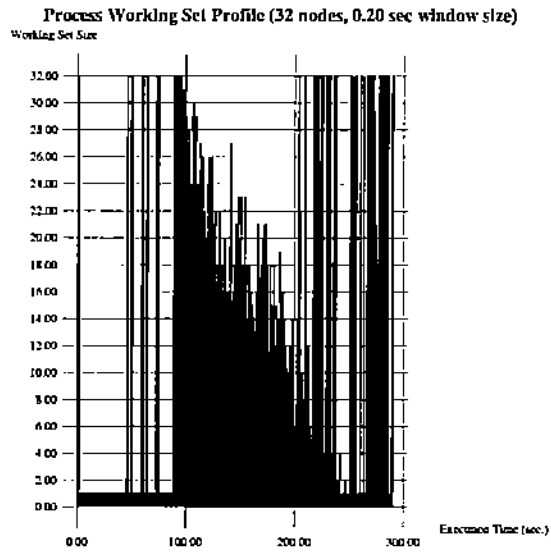


Figure 8. Envelope working set profile,  $\Delta = 0.2$  seconds, for a parallel execution using 32 nodes.

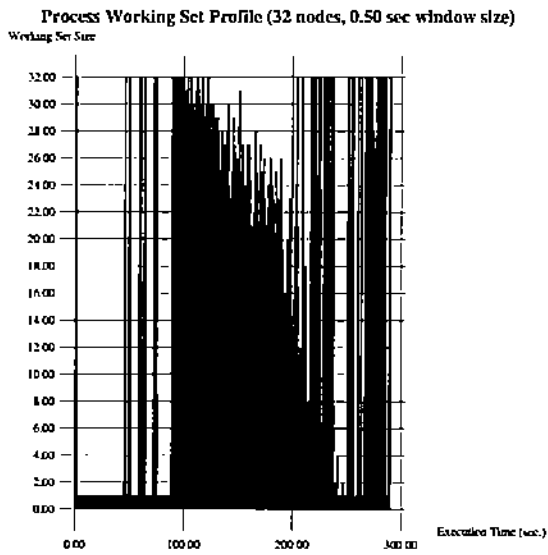


Figure 9. Envelope working set profile,  $\Delta = 0.5$  seconds, for a parallel execution using 32 nodes.

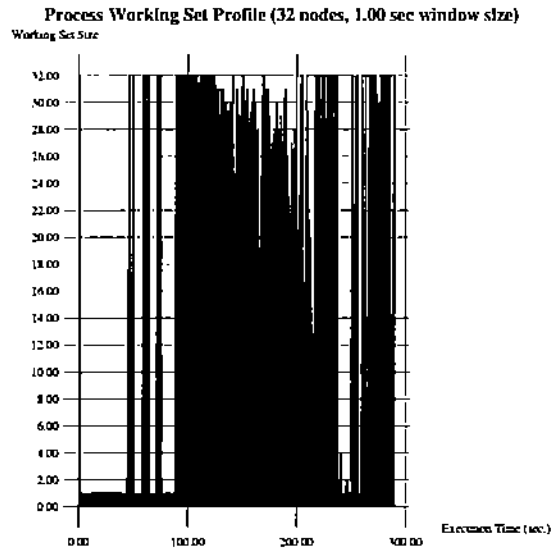


Figure 10. Envelope working set profile,  $\Delta = 1$  second, for a parallel execution using 32 nodes.

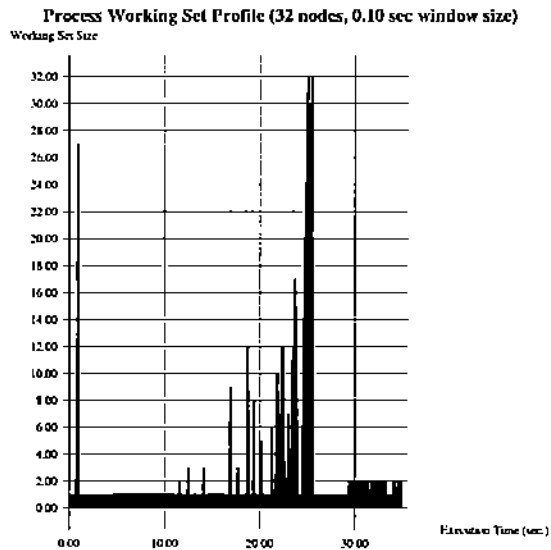


Figure 11. Fftsynth working set profile,  $\Delta = 0.1$  seconds, for a parallel execution using 32 nodes and 11MB of input data.

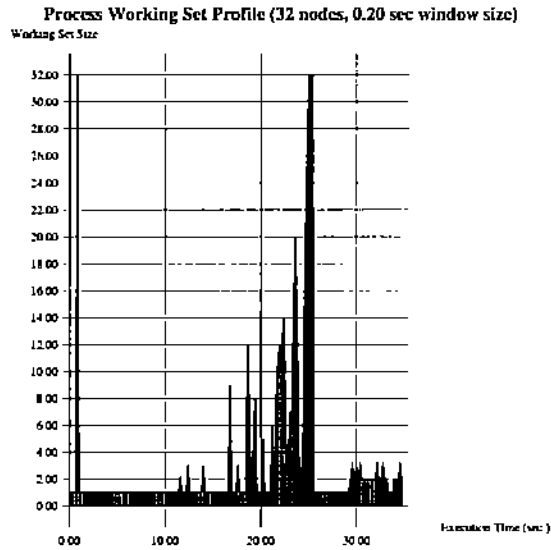


Figure 12. Fftsynth working set profile,  $\Delta = 0.2$  seconds, for a parallel execution using 32 nodes and 11MB of input data.

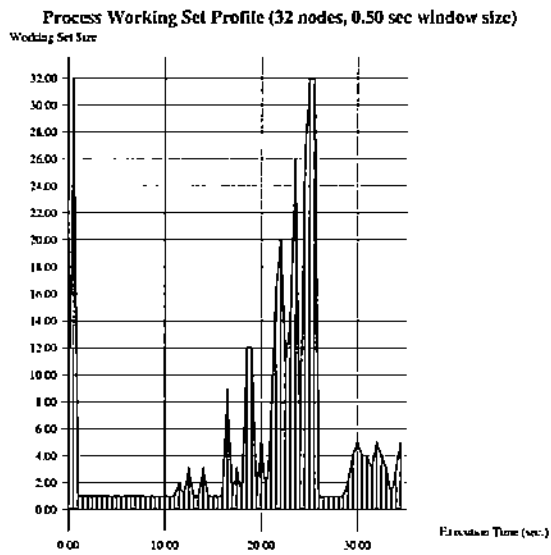


Figure 13. Fftsynth working set profile,  $\Delta = 0.5$  seconds, for a parallel execution using 32 nodes and 11MB of input data.

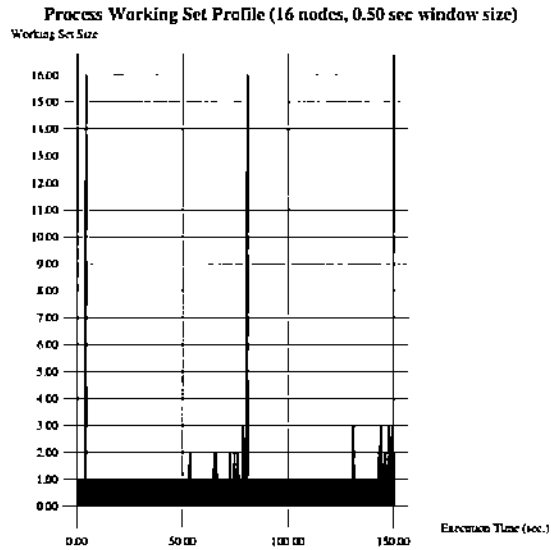


Figure 14. Fftsynth working set profile,  $\Delta = 0.5$  seconds, for a parallel execution using 16 nodes and 40MB of input data.

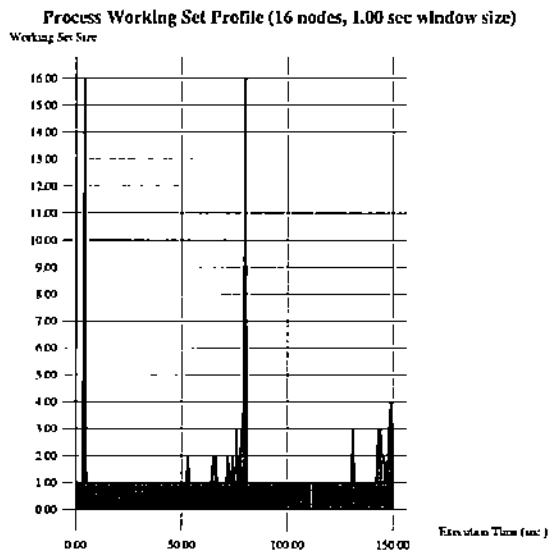


Figure 15. Fftsynth working set profile,  $\Delta = 1$  second for a parallel execution using 16 nodes and 40MB of input data.

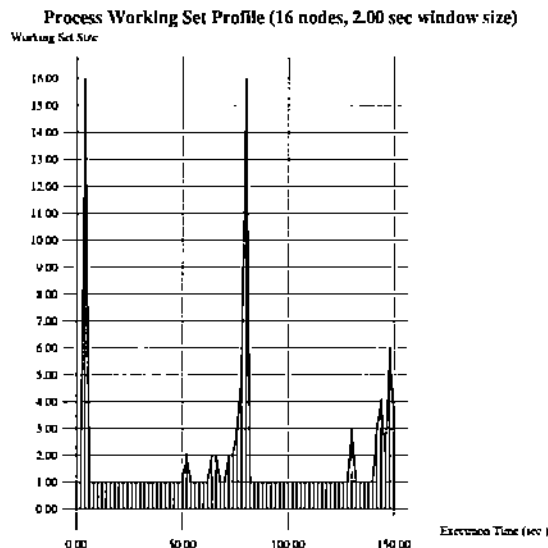


Figure 16. Fftsynth working set profile,  $\Delta = 2$  seconds, for a parallel execution using 16 nodes and 40MB of input data.

#### 4.4 Discussion of the results

The two programs we have examined exhibit very different communication patterns. The envelope program starts with a global communication stage followed by a relatively long period of isolation when different threads of control perform initialization tasks, and caching the input lattice across nodes. This phase ends after about 80 seconds and a period of intense communication follows until the execution completes. As expected, during this period the working set size is close to the size of the process group; different PEs use the active message facilities available on the Paragon to retrieve data stored elsewhere, as needed by the local computations.

Even for a relatively small window size (0.2 seconds) we observe that a process has a high probability of communicating with virtually all other processes in the group. This trend becomes dominant as the window size increases to 0.5 and 1.0 seconds. The same trend is observed whether 64 or 32 PEs are used.

The `fftsynth` exhibits quite an opposite behavior. The threads of control work in isolation until the time for the global exchange course. Again this behavior is independent upon the number of PEs.



## 5 Conclusions

We conjecture that sometimes processes belonging to a process group exhibit the following communication pattern: once they start communicating, they do so for some time; when they are working in isolation, they tend to maintain the state. Clearly, not all applications are expected to exhibit such a behavior at all times. Similarly, there are programs which do not exhibit locality of reference. But when process groups do exhibit locality of communication, the state of the process group provides information useful to hide the latency of I/O and paging for the individual processes in the group, and to avoid processor fragmentation. In the first case, when the process experiencing a high latency operation is temporarily independent, a local context switch to another temporarily independent process is better than busy waiting.

Processor fragmentation can be avoided, or its negative effects diminished through dynamic scheduling. Indeed, instead of being constrained to schedule all the members of a group, the system scheduler has the flexibility to schedule any number of processes larger than the kernel of  $n_p^C$  communicating processes at that time. Dynamic scheduling requires global decisions. The scheduler needs to maintain the state vectors for all process groups.

The analysis presented in Section 4 was done off line using data collected at run time. The clustering algorithm allows us to identify several clusters of processes in a process group communicating to each other. Yet, at run time, it is very expensive to identify such clusters. Using status vectors, all we know is the set of processes which are in a communication period,  $n_p^C$ . If we had the knowledge of the clusters of communicating processes, then the system (global) scheduler might be able to schedule one cluster of a process group at a time.

Last, but not least, we argue that checkpointing of a parallel program may be done more effectively using the information provided by the state vectors introduced in §3.1.

Finally, we need to answer the question of what happens if a process group does not exhibit locality of communication and the system attempts to either hide the latency by local context switches or to reduce the effects of processor fragmentation by dynamic scheduling. The answer is that the overall effect will be loss of performance through ill advised local or global context switches. But this is expected, as we already know that hierarchical storage systems have poor performance when programs do not exhibit locality of reference.

It is extremely difficult to assess the performance of dynamic scheduling without intensive experimentation but this class of algorithms have the potential of improved performance compared with existing algorithms.

Though it may seem a heresy at this time, we believe that economic realities will force the integration of parallel applications into mainstream computing.

## 6 Acknowledgments

The authors are indebted to the anonymous reviewers for their insightful comments which have contributed to improve the quality of this paper.

## References

- [1] M. J. Atallah, C. Lock, D. C. Marinescu, H. J. Siegel, and T. L. Casavant. Models and algorithms for co-scheduling compute-intensive tasks on a network of workstations. *Journal of Parallel and Distributed Computing*, 16:319–327, 1992.
- [2] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proceedings of Supercomputing '94*, November 1994.
- [3] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. *Performance Evaluation Review*, 22(1):33–44, May 1994.
- [4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. McGraw-Hill Book Company, 1991.
- [5] M. A. Cornea-Hasegan, D. C. Marinescu, and Z. Zhang. Data management for a class of iterative computations on distributed memory MIMD systems. *Concurrency: Practice and Experience*, 6(3):205–229, 1994.
- [6] M. A. Cornea-Hasegan, Z. Zhang, R. E. Lynch, D. C. Marinescu, A. Hadfield, J. K. Muckelbauer, S. Munshi, L. Tong, and M. G. Rossmann. Phase refinement and extension by means of non-crystallographic symmetry averaging using parallel computers. *Acta Crystallographica*, D51:749–759, 1995.
- [7] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

- [8] A. Ieumwananonthachai, A. N. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan. Intelligent mapping of communicating processes in distributed computing systems. In *Supercomputing '91*, pages 512–521, November 1991.
- [9] S. Majumdar, D. L. Eager, and R. B. Bunt. Characterization of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation* 13(2):109–130, 1991.
- [10] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd Intl. Conf. Distributed Computing Systems*, pages 22–30, October 1982.
- [11] K. C. Sevcik. Characterization of parallelism in applications and their use in scheduling. *Performance Evaluation Review*, 17:171–180, May 1989.
- [12] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2–3):107–140, March 1994.
- [13] Tumuri C., Mohan C.K., and Choudhary, A. Unsupervised algorithms for learning emerging spatio-temporal correlations. In *Proceedings IEEE-ICNN International Conference in Neural Networks*.
- [14] K. Y. Wang and D. C. Marinescu. Correlation of the paging activity of individual node programs in the SPMD execution mode. In *Proceedings of the 28th Hawaii International Conference on System Sciences, HICSS'28*, pages 61–71. IEEE Press, January 1995.
- [15] K. Y. Wang. Hiding the latency of paging and I/O operations on massively parallel systems. *Ph. D. Dissertation*, Computer Sciences Department, Purdue University, August 1996.
- [16] Maurice V. Wilkes. *Computing Perspectives*, Morgan Kaufmann, 1995.